SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 217–233 (1997)

Research

A Technique for the Selective Revalidation of OO Software

PEI HSIA,[1]* XIAOLIN LI,[1] DAVID CHENHO KUNG,[1] CHIH-TUNG HSU,[1] LIANG LI,[1] YASUFUMI TOYOSHIMA[2] AND CRIS CHEN[2]

[1]*Department of Computer Science and Engineering, The University of Texas at Arlington, P.O. Box 19015, Arlington, TX 76019-0015, U.S.A.*
[2]*Fujitsu Network Transmission Systems, Inc., 2540 First Street, #201, San Jose, CA 95131, U.S.A.*

SUMMARY

The object-orientated paradigm provides the power for software development but at the same time introduces some brand new problems. One of these problems is that the relationships among classes are more complex and difficult to identify than those in the traditional paradigm. This problem becomes a major obstacle for regression testing of OO software, in which the relationships among classes as well as those between test cases and classes, must be determined a priori. In this paper we propose a new method to select only a fraction of the test cases from the entire test suite to revalidate an OO software system. This method is based on the concepts of class firewall and of marking all the classes 'touched' by a test case. From the class firewall, we can identify all of the affected classes after a new version of software is released. Together with the markings, we can also identify all the test cases in the test suite that need to be retested after the software change. A step-by-step process is proposed to identify the relationships between classes and test cases, compute the class firewall, and select only the appropriate test cases for retesting. © **1997 by John Wiley & Sons, Ltd.**

J. Softw. Maint., **9**, 217–233 (1997)

No. of Figures: 4. No. of Tables: 6. No. of References: 17.

KEY WORDS: software validation; object-orientated software testing; regression testing; acceptance testing; class firewall; software maintenance

1. INTRODUCTION

Software revalidation involves retesting the modified software to ensure that the software behaves correctly after the modification. There are essentially four issues in software revalidation:

* Correspondence to: Pei Hsia, Department of Computer Science and Engineering, University of Texas at Arlington, PO Box 19015, Arlington, TX 76019-0015, U.S.A. E-mail: hsia@cse.uta.edu

Contract grant sponsor: Texas Advanced Technology Program; Contract grant number: 003656-097.
Contract grant sponsor: Fujitsu Network Transmission Systems, Inc.
Contract grant sponsor: IBM Center for Advanced Studies.
Contract grant sponsor: Software Engineering Center for Telecommunications.

CCC 1040–550X/97/040217–17$17.50

Received 21 October 1996
Revised 6 May 1997

1. change impact identification,
2. test suite maintenance,
3. test strategy, and
4. test case selection.

Resolving the first issue involves locating all the modules and other program segments that are affected by the modification. Test suite maintenance attempts to keep the test suite status current and reusable for future revalidation. This involves identifying and eliminating the obsolete test cases and adding new relevant test cases into the test suite. The third issue in regression testing is to find a testing sequence to conduct a cost-effective retesting of the software. The fourth issue, test case selection, involves choosing a subset of the existing test suite to re-test the software after a modification. The objective of test case selection is to achieve maximum coverage with minimal time, cost and effort.

Two major factors contribute to the cost of revalidation. The first factor is maintaining a complete and up-to-date test suite to cover every part of the software. This may require generating new test cases and deleting irrelevant test cases; however, identifying the obsolete test cases so that they can be eliminated from the test suite is rather difficult. The second factor is revalidating the modifications by rerunning an appropriate subset of the test suite. This involves selecting a relevant set of test cases and a test strategy.

Software revalidation is a complex and expensive activity. The introduction of the object-orientated ('OO') paradigm makes this issue even more complicated and difficult to deal with. The problem is due to the use of numerous OO features such as inheritance, messaging, polymorphism and dynamic binding. The current practice of OO revalidation is still in its infancy. The transfer of traditional testing strategies to OO applications is relatively difficult. For example, unlike traditional procedural-language software, a hierarchy of program control structure is not existent in OO software. This prevents the traditional integration testing strategies like top-down, bottom-up and sandwich approaches from being directly applied to OO software. The intricate relationships and interactions between classes make test case design difficult.

This paper is organized as follows. The next section reviews related research on the regression testing of OO software. The proposed test case selection approach is presented in Section 3. Section 4 presents the results of our experiments to illustrate the application and the effectiveness of this approach. The final section gives concluding remarks for future research in this domain.

2. RELATED WORK

A number of studies on the selective revalidation of procedural-language software have been conducted in the past. Harrold and Soffa (1988) proposed an approach for analysing the change effects within a module. They employed a dataflow graph to identify the affected definition-use pairs and/or paths. Testing effort is reduced by retesting only the affected definition-use paths and the new paths. They extended the approach to identify affected modules at the inter-module level (Harrold and Soffa, 1989). Different approaches using a control-flow graph to identify the affected control-flow paths in a module include Prather and Myers (1987) and Laski and Szermer (1992).

Leung and White (1990) introduced the firewall concept to enclose all modules affected by a module modification. They based the identification of the firewall on a call graph. By retesting only the modules that are within the scope of the firewall, test effort can be saved. The firewall concept was later extended to include a data-orientated firewall to enclose the affected modules that are global-data-related to the modified module (White and Leung, 1992). Fischer (1977, 1980) employed the concept of a zero–one integer programming model to find a minimum set of test cases which cover one of the path criteria in unit regression testing. Hartmann and Robson (1989, 1991) extended Fischer's method to investigate other retest criteria and constraints such as retesting at minimum cost, time or effort.

Lee and He (1990) applied the zero–one programming model on a test matrix to minimize testing effort in functional regression testing. Yau and Kishimoto (1987) proposed a selective retesting method based on the concept of input-partitioning. The idea is to partition a program's input domain into several classes and combine the input data to traverse various paths through the code. A minimum set of test cases can be derived by identifying all possible paths that reach the modified code. Leung and White (1988) used the concept of function table to relate the functions and their associated test cases to maintain the test suite. Test cases are classified into reusable, obsolete and retestable so that only the retestable and new test cases are considered as tests in the new regression testing.

Although extensive efforts have been given to the research on traditional software revalidation, the issue of selective revalidation on object-orientated software as yet is not well addressed. Rothermel and Harrold (1994) presented an algorithm to construct dependence graphs for classes and application programs. The dependence graphs are used to determine which tests in an existing test suite can cause a modified class or program to produce a different output from the original.

The approach described in this paper addresses the issue of test case selection and emphasizes the detection of the run-time relationships between object classes and test cases. Our approach can handle not only the class inheritance relationship, but also the aggregation and association relationships. Kung *et al*. (1994, 1995) introduced a concept of class firewall to identify the effect of a class-level modification. A test order generation algorithm was also proposed to help retest the classes within the class firewall. The approach described in this paper applies the concept of class firewall to test case selection in the revalidation process.

3. TEST CASE SELECTION FOR REVALIDATION

3.1. Three steps

To make the OO software revalidation process effective and cost-efficient, we propose an approach in which the revalidation activities are performed in three steps:

1. Determining the relationship between class and test case. Appropriate probes are added to program code to detect the existence of possible relationships between the

use of a class and the execution of a test case. Inserting probes into the program facilitates the detection of the relationships between classes and test cases.
2. Identifying the change impacts of a class. An object relation diagram (ORD) is constructed from the source code to describe the static relationships among the classes. The ORD will be used as the basis for evaluating the cascading class change effects.
3. Selecting test cases for the revalidation. Some test cases in the original test suite have to be retested to ensure the correctness of the desired functionality of the OO software, while certain test cases may be ruled out in the revalidation process because they have no direct or indirect relationship with the changed class.

An OO program may be tested at different levels. In this paper we focus on the selective revalidation at the class level. Our approach can be used to evaluate the change impacts in a class library and to identify the test cases affected by the class changes.

3.2. Determining the relationship between class and test case

The test cases represent the possible inputs to the software from the system environment. In this paper we consider only those test cases that are visible to the end users. Our approach requires that a test log file be created to keep track of the classes that have been traversed by the test cases when the software system undergoes the acceptance testing. To keep track of the classes traversed by a specific test case, a special probe function MARK is inserted into appropriate position within the source code (see Figure 1). This preprocessing step paves the way for identifying the dynamic relationship between the classes and the test cases.

During the preprocessing phase, a script is used to extract the class name for each class in the OO software, and a probe function is then added to each class constructor and each member function in the class. Whenever a class object is declared or accessed by the member functions, the statements in the constructor or member function body will be automatically executed, thus triggering the incorporated probe function.

Definition

For each test case T, a touch set $TS(T)$ is defined as the set of the classes traversed by the execution of T.

Note that a touch set is determined at run-time. From a touch set $TS(T)$, a tester can distinguish those classes that have been used during the execution of T from the classes

```
MARK(classname)
    write the class name to the test log file
    write the test case id to the test log file
```

Figure 1. The probe function

that have not been used at all. Although the probe function MARK (see Figure 1) is added to each of the class constructors and member function bodies, MARK may or may not be triggered by a test case. If a class does not declare any constructor, the script will insert a constructor with the probe function MARK inside.

The probe function will not be triggered if no test cases use the class containing the probe function. Therefore, using the strategy of inserting the MARK function enables one to capture the dynamic relationships between classes and test cases. The dynamic features of these relationships are often difficult to detect and describe by the static program analytical techniques (e.g., tracing the program control flows manually).

3.3. Identifying the impacts of changing a class

The identification of class change impacts is a two-step process. The first step is the construction of an object-relation diagram (ORD) that identifies the relationship between the classes in the OO software. The ORD then serves as the basis upon which the task in the second step, called class firewall computation, is carried out (Kung *et al.*, 1995). The firewall for a changed class includes those classes that have been affected by the changes in a class. An object relation diagram represents the relationship of inheritance, aggregation and association among classes. An inheritance relation between two classes means that the properties defined for an object class are automatically defined for all of its subclasses, except where selective or overriding inheritance is specified. By using an aggregation relation, a composite object can be defined on the basis of its component objects, and the composite object is called the aggregated class object. The association relation encompasses a broader range of relationship links such as data dependence control dependence, or message passing between two independent object classes.

Definition

The ORD for an OO program Q is a directed graph $ORD = (V, E)$, where V is a set of nodes representing the object classes in Q, and $E \subseteq V \times V$ is a set of edges which describe the inheritance, aggregation or association relationships between classes. An edge from class A to class B indicates that A is dependent on B.

More specifically, $\langle C_1, C_2 \rangle, \in E$ if and only if one of the following conditions holds:

- C_1 is a derived class of C_2,
- C_1 is an aggregate class of C_2 (i.e., C_2 is part of C_1), or
- C_1 is associated with C_2 either by accessing its data members or by passing some messages.

It can be seen that the binary relation E is transitive; that is, if class B depends on class A and class C in turn depends on C, then C also depends on A. Figure 2 shows a portion of the object relation diagram derived from an ATM simulation program. Class TRANSACTION is determined to have an association relationship with class customer because customer is a parameter of member function TRANSACTION( ). Since the


```
class customer
{ char *userId;
  ... }

class TRANSACTION
{ TRANSACTION(customer*)...}

class TRANSFER_TRAN:public TRANSACTION
{ ... }

class MENU
{ TRANSFER_TRAN  * t1 }
```

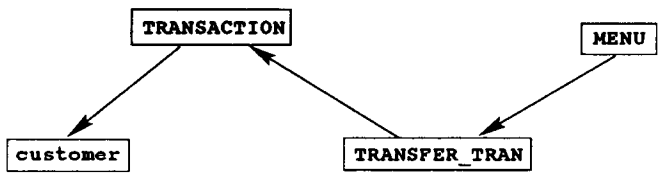


Figure 2. Obtaining class dependency relationships from source code

objects of class TRANSFER_TRAN become data members in class MENU, there exists an aggregation relationship between class TRANSFER_TRAN and class MENU. The inheritance relationship between classes TRANSACTION and TRANSFER_TRAN is straightforward.

Any changes made in an OO program will make it necessary for users to redo acceptance testing. The various changes in an OO program can be classified into three categories:

(1) changes that affect the object behaviour represented by the states of the data members;
(2) changes that affect the operations and behaviour of a class's member functions; and
(3) changes that affect the relationships and dependencies between a class and other classes.

In this paper, we consider only the first two types of changes and do not deal with class relation changes.

Definition

The class firewall for a class *C*, denoted *CFW*(*C*), is defined as a set of classes that are dependent on *C* as described by an *ORD*.

Intuitively, a class firewall for a class *C* is the set of classes that may be affected by changes to class *C*. Figure 3 shows a class firewall of the example in Figure 2.

3.4. Test case selection

The changes made to a class in an OO program not only have impacts on other classes but also influence the set of test cases needing to be retested. The problem of identifying the test cases affected by class change is complicated by the fact that the OO paradigm supports a variety of class dependence relationships. Our study indicates that the class change impacts on the test cases can be detected and evaluated on the basis of class

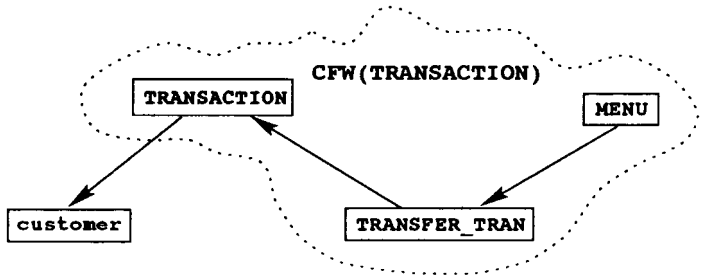


Figure 3. A class firewall

firewall computation. Intuitively, if a programmer makes a change to class C, testers only need to select test case T such that the touch set $TS(T)$ and class firewall $CFW(C)$ have a non-empty intersection.

Our approach uses matrix arithmetic to locate the test cases needed for retesting. Specifically, we define two matrices:

1. a $1 \times n$ class impact matrix M_c to indicate which classes have been included in a class firewall, where n is the total number of classes in the OO program under test; and
2. an $n \times k$ dependence matrix M_d to show the relationships between classes and test cases, where k is the number of test cases in the old test suite.

Once M_c and M_d are defined, the test cases needed for retesting can be determined from a matrix product $M_c \times M_d$. A zero-valued element q_{1i} in $M_c \times M_d$ implies that test case T_i has not been affected by the class changes, whereas each non-zero element q_{1i} indicates that test case T_i may be affected by class changes and therefore should be included in the new test suite for the revalidation.

4. EXPERIMENTS

4.1. ATM simulation software

4.1.1. Objective

This is a Unix-based simulation program written in C++. It consists of six program files and defines 13 object classes. Since the functionalities and behaviours of automated teller machines (ATMs) are well-known, we only give a brief description of each class in the software and assume the software requirements are self-explanatory. The main focus of this ATM experiment is to determine the feasibility of our approach. Specifically, we want to know:

1. whether our OO testing tool can identify the relationships among these 13 classes and perform class firewall computation; and

2. whether the number of test cases can be reduced for the revalidation.

Table 1 lists the classes defined in the software.

The relationships among the classes are identified by our OO testing tool, as illustrated in Figure 4. For example, class MENU turns out to be a subclass of TEXT_DISPLAY, and there exists an aggregation relationship between class TRANSFER_TRAN and class customer. Initially we tested the simulation software with 12 test cases, each testing a special aspect of the software:

T_1: Input invalid user ID, exit.
T_2: Valid user ID, invalid PIN, exit.
T_3: Valid user ID, valid PIN, exit.
T_4: Test transactions of deposition, withdraw, transfer, inquiry, valid input.
T_5: Test the withdraw of money, invalid input.
T_6: Test the deposition of money, invalid input.
T_7: Test the inquiry, invalid input.
T_8: Test the transfer of money, invalid input.
T_9: Input SuperUserID is wrong and exit.
T_{10}: Input SuperUserID is correct, but PIN is wrong and exit.
T_{11}: Test the SuperUser function, valid input.
T_{12}: Test the SuperUser function, invalid input.

4.1.2. Step 1: determining the relationships between class and test cases

A probe function MARK is automatically inserted into the constructor of each class. If there is no explicit constructor, a constructor with a MARK function is inserted. The following is a list of touch sets identified by our testing tool. From these touch sets, a tester can determine which classes are actually used by a certain test case. For example,

Table 1. Classes in ATM simulation software

Class	Class name	Class description
C_1	account	Maintain user account information
C_2	TEXT_DISPLAY	Print out screen messages
C_3	customer	Maintain customer information
C_4	TRANSACTION	Activate a transaction
C_5	WITHDRAW_TRAN	Withdraw money from an account
C_6	DEPOSIT_TRAN	Deposit money into an account
C_7	INQUIRY_TRAN	Retrieve relevant account information
C_8	TRANSFER_TRAN	Transfer money to another account
C_9	MENU	Maintain screen display formats
C_{10}	MSG	Maintain message formats
C_{11}	BAL_MSG	Prepare a message for a balance report
C_{12}	STR	Perform string operations
C_{13}	ATM	Maintain ATM information

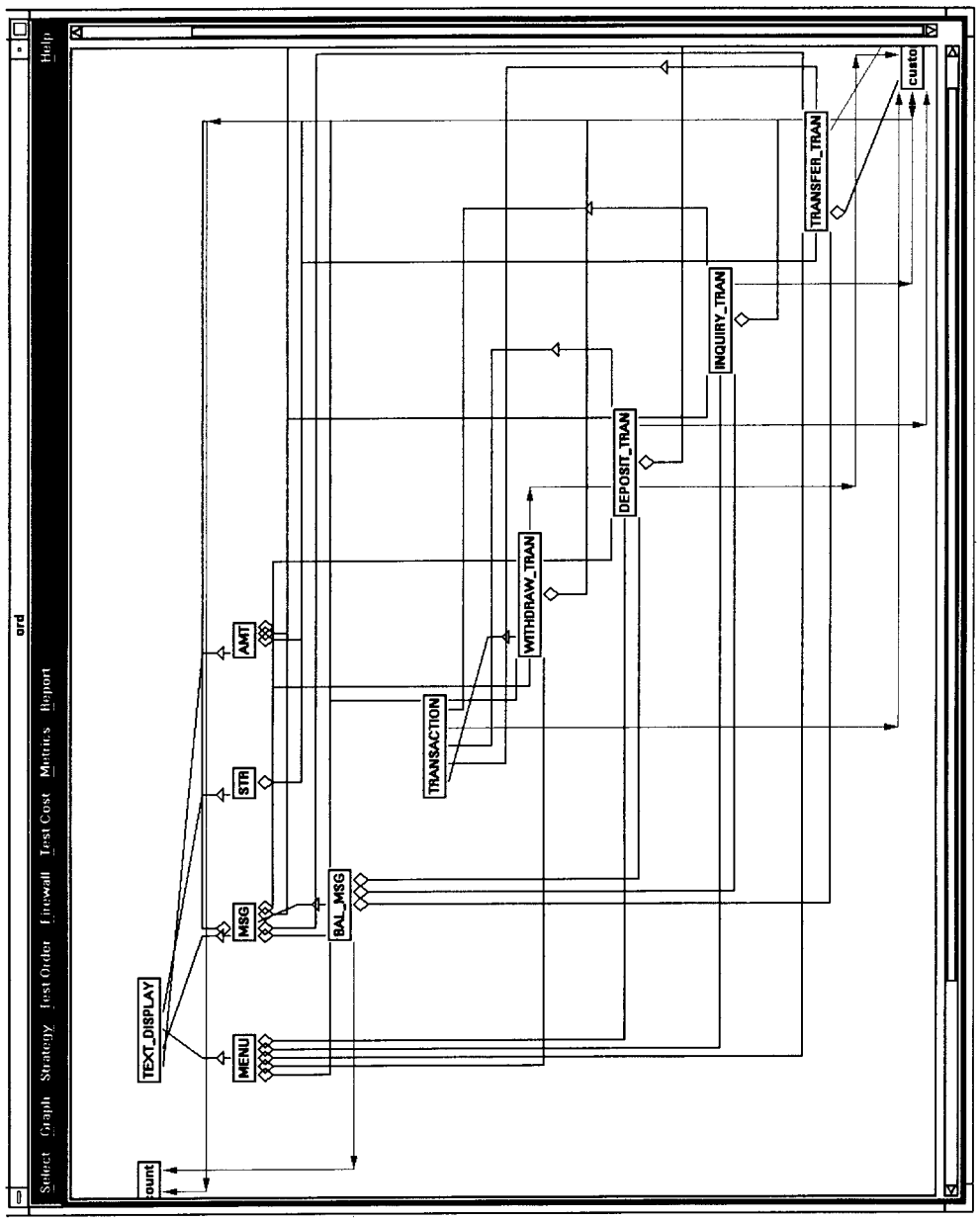


Figure 4. An ORD screen for the ATM experiment

classes TEXT_DISPLAY, MSG, and STR are determined to be used by test case T_1 during its execution.

$TS(T_1) = \{C_2, C_{10}, C_{12}\}$
$TS(T_2) = \{C_1, C_2, C_{10}, C_{12}\}$
$TS(T_3) = \{C_1, C_2, C_9, C_{10}, C_{12}\}$
$TS(T_4) = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_{10}, C_{11}, C_{12}, C_{13}\}$
$TS(T_5) = \{C_1, C_2, C_3, C_4, C_5, C_9, C_{10}, C_{12}, C_{13}\}$
$TS(T_6) = \{C_1, C_2, C_3, C_4, C_6, C_9, C_{10}, C_{12}, C_{13}\}$
$TS(T_7) = \{C_1, C_2, C_3, C_4, C_7, C_9, C_{10}, C_{11}, C_{12}, C_{13}\}$
$TS(T_8) = \{C_1, C_2, C_3, C_4, C_8, C_9, C_{10}, C_{11}, C_{12}\}$
$TS(T_9) = \{C_2, C_{10}, C_{11}\}$
$TS(T_{10}) = \{C_2, C_{10}, C_{11}\}$
$TS(T_{11}) = \{C_2, C_{10}, C_{11}, C_{12}, C_{13}\}$
$TS(T_{12}) = \{C_2, C_{10}, C_{11}, C_{12}, C_{13}\}$

4.1.3. Step 2: identifying the impacts of changing a class

Assume that class WITHDRAW_TRAN (i.e., C_5) has to be changed to enhance the ATM functionalities during system maintenance. From the object relation diagram (ORD) in Figure 4, our testing tool determines the class firewall for C_5 as $CFW(C_5) = \{C_3, C_4, C_5, C_6, C_7, C_8\}$. $CFW(C_5)$ clearly indicates that the changes in one class can influence other classes in an OO program. It shows that changes in class WITHDRAW_TRAN will affect six classes, leaving the other seven classes unaffected. The information on the class change impacts can then be used as a guide for the test case selection in the next step.

4.1.4. Step 3: test case selection

From $CFW(C_5)$ we can see that six classes are included in the class firewall. The corresponding class impact matrix P is then determined to be:

$$M_c = [\ 0,\ 0,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0\ ] \tag{1}$$

The dependence matrix M_d can be constructed from the touch sets $TS(T_1) \sim TS(T_{12})$ as mentioned in Step 1, is shown in Table 2, and hence:

$$M_c \times M_d = [\ 0,\ 0,\ 0,\ 6,\ 3,\ 3,\ 3,\ 3,\ 0,\ 0,\ 0,\ 0\ ] \tag{2}$$

The non-zero elements in columns 4, 5, 6, 7 and 8 indicate that only test cases T_4, T_5, T_6, T_7, and T_8 are required to retest the software if class WITHDRAW_TRAN is changed after the initial round of testing. The other seven test cases do not have to be retested.

4.2. Interactive graphic-drawing software

The second software package used in our experiment is a PC-based drawing program (Priestley, 1996). The package consists of 11 program files and 18 header files, which

Table 2. Dependence matrix for ATM simulation software experiment

Class	Test cases											
	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
account	0	1	1	1	1	1	1	1	0	0	0	0
TEXT_DISPLAY	1	1	1	1	1	1	1	1	1	1	1	1
customer	0	0	0	1	1	1	1	1	0	0	0	0
TRANSACTION	0	0	0	1	1	1	1	1	0	0	0	0
WITHDRAWN_TRAN	0	0	0	1	1	0	0	0	0	0	0	0
DEPOSIT_TRAN	0	0	0	1	0	1	0	0	0	0	0	0
INQUIRY_TRAN	0	0	0	1	0	0	1	0	0	0	0	0
TRANSFER_TRAN	0	0	0	1	0	0	0	1	0	0	0	0
MENU	0	0	1	1	1	1	1	1	0	0	0	0
MSG	1	1	1	1	1	1	1	1	1	1	1	1
BAL_MSG	0	0	0	1	0	0	1	1	1	1	1	1
STR	1	1	1	1	1	1	1	1	0	0	1	1
ATM	0	0	0	1	1	1	1	0	0	0	1	1

are compiled and built into executables by using Borland Turbo C++. The software defines 26 object classes to support various figure-drawing functions, such as rectangle selection, ellipse drawing, and graphic element resizing and moving. Since the results from the first experiment indicate that our approach does have the potential to reduce regression testing cost, in this second experiment we are more interested in the overall performance of the approach. Particularly, we want to know (1) the effectiveness of the approach in the average case, and (2) which factors influence the effectiveness.

Table 3 lists all the classes used in the software, where C is a class name, $|CFW(C)|$ is the number of classes in the corresponding class firewall, and α is the ratio of $|CFW(C)|$ to the total number of classes (i.e., $|CFW(C)| \div 26$). Note that $|CFW(C)| = 1$ implies that there is no other class dependent on C, and therefore the changes only affect class C itself. Our initial test suite consists of 14 test cases, as listed in Table 4.

Having identified the class firewalls and developed the initial test suite, we want to examine the impacts of class changes on the test cases. The dependence matrix M_d of this experiment is shown in Table 5. Each column in the matrix represents the touch set of the related test case. A sparse matrix such as this is a clear indication that, in general, a test case uses only a small number of classes defined in the software.

For each of the 26 classes, our testing tool derives the corresponding class impact matrix M_c and computes $M_c \times M_d$. The experiment results are listed in Table 6, where C is the changed class, T' is the set of test cases affected by the changes, $|T'|$ is the size of T', and p is the ratio of $|T'|$ to the total number of test cases in the initial test suite. Recall that owing to the dependence relationships of classes, it may well be the case that more test cases are affected by changes in a class than those directly using the class. For example, all 14 test cases are affected by the changes in class Node, despite the fact that the class is used directly by three test cases only (i.e., T_5, T_6 and T_7).

It can be seen from Table 6 that in the best case the changes made to a class (e.g.,

Table 3. Classes and their firewall sizes

C	$\lvert CFW(C)\rvert$	α
ControlPoint	16	0·62
Collection	10	0·38
Node	11	0·42
Display	21	0·81
Drawing	8	0·31
Rectangle	3	0·12
Ellipse	3	0·12
GraphicsScreen	1	0·04
Mouse	4	0·15
Poller	1	0·04
CreationTool	5	0·19
LineTool	2	0·08
SelectionTool	2	0·08
Application	4	0·15
CollectionEditor	1	0·04
DiagramEditor	1	0·04
DOSKeyboard	1	0·04
Element	15	0·58
Line	3	0·12
EventHandler	2	0·08
Keyboard	3	0·12
MSMouse	1	0·04
Tool	7	0·27
RectangleTool	2	0·08
EllipseTool	2	0·08
ToolManager	8	0·31

Table 4. Test cases in the initial test suite

Test case	Description
T_1	Initiate the program and then exit
T_2	Select the rectangle-drawing operation
T_3	Select the ellipse-drawing operation
T_4	Select the line-drawing operation
T_5	Draw a rectangle
T_6	Draw an ellipse
T_7	Draw a line
T_8	Initiate the select-element operation
T_9	Cancel the select-element operation
T_{10}	Select an element (a rectangle, an ellipse or a line)
T_{11}	Move an element
T_{12}	Unselect an element
T_{13}	Initiate select-element operation and drag (outside of any element)
T_{14}	Resize an element (click on handle and drag)

Table 5. Dependence matrix for graphic-drawing software

Class	Test case													
	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
Application	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Collection	1	0	0	0	0	0	0	1	0	0	0	0	0	0
CollectionIter.	0	0	0	0	0	0	0	0	1	1	1	1	1	1
Node	0	0	0	0	1	1	1	0	0	0	0	0	0	0
ControlPoint	0	0	0	0	1	1	1	0	0	0	0	0	0	0
DiagramEditor	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Display	1	0	0	0	0	0	0	0	0	0	0	0	0	0
DOSKeyboard	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Drawing	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Element	0	0	0	0	1	1	1	0	0	0	0	0	0	0
Rectangle	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Line	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Ellipse	0	0	0	0	0	1	0	0	0	0	0	0	0	0
EventHandler	1	0	0	0	0	0	0	0	0	0	0	0	0	0
GraphicScreen	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Keyboard	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Mouse	1	0	0	0	0	0	0	0	0	0	0	0	0	0
MSMouse	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Poller	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Tool	0	1	1	1	0	0	0	1	0	0	0	0	0	0
CreationTool	0	1	1	1	0	0	0	0	0	0	0	0	0	0
RectangleTool	0	1	0	0	0	0	0	0	0	0	0	0	0	0
LineTool	0	0	0	1	0	0	0	0	0	0	0	0	0	0
EllipseTool	0	0	1	0	0	0	0	0	0	0	0	0	0	0
SelectionTool	0	0	0	0	0	0	0	1	0	0	0	0	0	0
ToolManager	1	1	0	0	0	0	0	0	0	0	0	0	0	0

class DiagramEditor) influence only one test case, implying that 13 other test cases are not required for the revalidation. However, the table also shows that the changes in four of the classes affect the initial test suite so seriously that all of the 14 test cases must be retested. A better measurement for the overall performance is to look at the average case. If we assume that each class is equally likely to be changed, on average $|T'|$ is equal to 4·54. This indicates that on average only 4·54 out of 14 test cases will be affected by the changes in a class. In other words, we can expect that the approach will provide a savings of 67·6% for the software revalidation.

5. CONCLUDING REMARKS

We have described a selective revalidation technique for OO software. The proposed approach is based on the ideas of object relation diagram, class firewall, class-level probe function and a test-class matrix. The experiments show that our approach can save time and cost. First, the time of applying our approach is minimal because the entire revalidation

Table 6. Test cases affected by class changes

C	T'	$\|T'\|$	p
ControlPoint	$T_1 \sim T_{14}$	14	1·000
Collection	$T_1 \sim T_4$, $T_8 \sim T_{14}$	11	0·785
Node	$T_1 \sim T_{14}$	14	1·000
Display	$T_1 \sim T_{14}$	14	1·000
Drawing	$T_1 \sim T_4$, T_8	5	0·357
Rectangle	T_1, T_2, T_5	3	0·214
Ellipse	T_1, T_3, T_6	3	0·214
GraphicsScreen	T_1	1	0.071
Mouse	T_1	1	0·071
Poller	T_1	1	0·071
CreationTool	$T_1 \sim T_4$	4	0·286
LineTool	T_1, T_4	2	0·143
SelectionTool	T_1, T_8	2	0·143
Application	T_1	1	0·071
CollectionEditor	$T_9 \sim T_{14}$	6	0·429
DiagramEditor	T_1	1	0·071
DOSKeyboard	T_1	1	0·071
Element	$T_1 \sim T_{14}$	14	1·000
Line	T_1, T_4, T_7	3	0·214
EventHandler	T_1	1	0·071
Keyboard	T_1	1	0·071
MSMouse	T_1	1	0·071
Tool	$T_1 \sim T_4$, T_8	3	0·357
RectangleTool	T_1, T_2	2	0·143
EllipseTool	T_1, T_3	2	0·143
ToolManager	$T_1 \sim T_4$, T_8	3	0·357

process—change impact identification and test case selection—has been fully automated. Second, the cost of revalidation can be significantly reduced if the number of selected test cases is well below the total number of test cases in the test suite. The cost saving is nearly linearly proportional to the number of unselected test cases.

Future research in this area should be directed at extending the current class-level revalidation technique to the member function and data member levels. Our approach assumes that each data member is private. That is, data member access from outside a class will invoke the class's constructor from which the probe function is executed. The use of the probe function should be expanded to encompass the cases of friend functions, and public data members and member functions.

Acknowledgements

The material printed in this paper is based on the work supported by the Texas Advanced Technology Program (Grant number 003656-097), Fujitsu Network Transmission Systems, Inc., the IBM Fellowship grant from IBM Center for Advanced Studies, Toronto, Canada, and the Software Engineering Center for Telecommunications at UTA.

References

Fischer, K. F. (1977) 'A test case selection method for the validation of software maintenance modifications', in *Proceedings COMPSAC77*, Chicago, IL, IEEE Computer Society Press, Los Alamitos, CA, pp. 421–426.

Fischer, K. F. (1980) 'A graph theoretic approach to the validation of software maintenance modifications', Doctoral dissertation, University of California, Los Angeles, CA, 141 pp.

Harrold, M. J. and Soffa, M. L. (1988) 'An incremental approach to unit testing', in *Proceedings Conference on Software Maintenance—1988*, IEEE Computer Society Press, Los Alamitos, CA, pp. 362–367.

Harrold, M. J. and Soffa, M. L. (1989) 'Interprocedural data flow testing', in *Proceedings of the Third Testing, Analysis and Verification Symposium*, ACM Press, New York, NY, pp. 158–167.

Hartmann, J. and Robson, D. J. (1989) 'Revalidation during the software maintenance phase', in *Proceedings Conference on Software Maintenance—1989*, IEEE Computer Society Press, Los Alamitos, CA, pp. 70–80.

Hartmann, J. and Robson, D. J. (1991) 'Techniques for selective revalidation', *IEEE Software*, **8**(1), 31–36.

Kung, D. C., Gao, J., Hsia, P., Wen, F., Toyoshima, Y. and Chen, C. (1994) 'Change impact identification in object-oriented software', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 202–211.

Kung, D. C., Gao, J., Hsia, P., Wen, F., Toyoshima, Y. and Chen, C. (1995) 'Class firewall, test order, and regression testing of object-oriented programs', *Journal of Object-Oriented Programming*, **8**(2), 51–56.

Laski, J. and Szermer, W. (1992) 'Identification of program modification and its applications in software maintenance', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 282–290.

Lee, J. A. N. and He, X. (1990) 'A methodology for test selection', *Journal of Systems and Software*, **13**(3), 177–185.

Leung, H. K. N. and White, L. (1988) 'A study of regression testing', Technical Report TR 88-15, Computing Science Department, University of Alberta, Edmondon, Canada, 16 pp.

Leung, H. K. N. and White, L. (1990) 'Insights into testing and regression testing global variables', *Journal of Software Maintenance*, **2**(4), 209–222.

Priestley, M. (1996) *Practical Object-Oriented Design*, McGraw-Hill Book Co., New York, NY, 350 pp.

Prather, R. E. and J.P. Myers Jr., J. P. (1987) 'The path prefix software testing strategy', *Transactions on Software Engineering*, **SE-13**(7), 761–765.

Rothermel, G. and Harrold, M. J. (1994) 'Selecting regression tests for object-oriented software', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society, Los Alamitos, CA, pp. 14–25.

White, L. and Leung, H. K. N. (1992) 'A firewall concept for both control-flow and data-flow in regression integration testing', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 262–271.

Yau, S. S. and Kishimoto, Z. (1987) 'A method for revalidating modified programs in the maintenance phase', in *Proceedings COMPSAC87*, IEEE Computer Society Press, Los Alamitos, CA, pp. 272–277.

Authors' biographies:

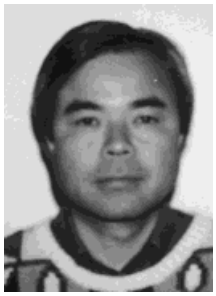
Pei Hsia is a Professor of Computer Science and Engineering at the University of Texas at Arlington. He is also the Director of the Software Engineering Center for Telecommunications. His research interests include requirements engineering, concurrent software engineering, incremental delivery and software testing. His e-mail address is: hsia@cse.uta.edu

Xiaolin Li received his B.S. degree in computer science in China, and his M.S. from Moorhead State University in Moorhead, Minnesota. Currently, he is a Ph.D. candidate in computer science at the University of Texas at Arlington. His research topic is object-orientated software testing.

David C. Kung is an Associate Professor of Computer Science and Engineering at the University of Texas at Arlington. He received his M.S. and Ph.D. degrees in computer science from the Norwegian Institute of Technology, and in 1990, he worked as a staff software scientist at International Software Systems, Inc. His research interests are in real-time systems and object-orientated systems.

Chih-tung Hsu received his B.A. and M.S. degrees in engineering in Taiwan, and an M.S. in computer science from the University of Texas at Arlington, where he is currently a Ph.D. candidate. His research interests include software requirements specification, object-orientated software testing, incremental delivery and concurrent software engineering.

Liang Li is a Senior Software Engineer in the Motorola Advanced Messaging Group, where he has worked on two-way and one-way paging systems. He earned an M.E. degree in engineering in China, and an M.S. in computer science from the University of Texas at Arlington. His research interests are in software management and object-orientated design patterns and object-orientated testing.



Yasufumi Toyoshima is Vice President of Fujitsu Network Communications, Inc. He received a B.S. degree in electrical engineering from the University of Osaka. His research interests include software engineering for object-orientated technology and software process management.

Cris Chen is a Senior Director at Fujitsu Network Communications, Inc. He holds a B.S. degree from Soochow University in Taiwan, and an M.S. in computer science from Santa Clara University in California. His research interests include software engineering methods and object-orientated technology and testing techniques.

